A VMM Security Kernel for the VAX Architecture

Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn*

Digital Equipment Corporation
Secure Systems Development
85 Swanson Road (BXB1-1/D03)
Boxborough, MA 01719-1326

Abstract

This paper describes the development of a virtual-machine monitor (VMM) security kernel for the VAX architecture. The paper particularly focuses on how the system's hardware, microcode, and software are aimed at meeting A1-level security requirements while maintaining the standard interfaces and applications of the VMS and ULTRIX-32 operating systems. The VAX security kernel supports multiple concurrent virtual machines on a single VAX system, providing isolation and controlled sharing of sensitive data. Rigorous engineering standards were applied during development to comply with the assurance requirements for verification and configuration management. The VAX security kernel has been developed with a heavy emphasis on performance and on system management tools. The kernel performs sufficiently well that all of its development is now carried out in virtual machines running on the kernel itself, rather than in a conventional time-sharing system.

1 Introduction

The VAX security kernel project is a research effort to determine what is required to build a production-quality security kernel, capable of receiving an A1 rating from the National Computer Security Center. A production-quality security kernel is very different from the many research-quality security kernels that have been built in the past, and this research effort has been primarily aimed at identifying the differences and their cost in development effort and in kernel complexity.

This paper describes how the VAX security kernel meets its five major goals:

- Meet all A1 security requirements.
- Run on commercial hardware without special modifications other than microcode changes for virtualization.
- Provide software compatibility for applications written for both the VMS and ULTRIX 32 operating systems.
- Provide an acceptable level of performance.
- Meet the requirements of a commercial software product.

The VAX security kernel is a research effort. Digital Equipment Corporation makes no commitment to offer it as a product.

2 Kernel Overview

The VAX security kernel is a virtual-machine monitor that runs on the VAX 8530, 8550, 8700, 8800, and 8810 processors.[1] It creates isolated virtual VAX processors, each of which can run either the VMS or ULTRIX-32 operating system. If desired, virtual machines running each of the operating systems can run simultaneously on the same computer system.[2] The VAX architecture was not virtualizable, and therefore extensions were made to the architecture and to the processor microcode to support virtualization. (See Section 3.2.)

Figure 1 shows a typical VAX security kernel configuration. While the VAX security kernel is a VMM, it is primarily a security kernel. Therefore, certain features traditionally seen in VMMs, such as self-virtualization or debugging of one VM from another, have been omitted to reduce kernel complexity.

*This paper presents the opinions of its authors, which are not necessarily those of the Digital Equipment Corporation. Opinions expressed in this paper must not be construed to imply any product commitment on the part of the Digital Equipment Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Ina Jo is a registered trademark of UNISYS Corporation.

MS-DOS is a registered trademark of Microsoft Corporation.

UNIX is a registered trademark of American Telephone and Telegraph Company.

The following are trademarks of Digital Equipment Corporation: DEC, DEC/CMS, DEC/MMS, PDP, PDP 11, ULTRIX, ULTRIX 32, VAX, VAX-11/730, VAX 8530, VAX 8550, VAX 8700, VAX 8800, VAX 8810, VAX 8820, VAX 8830, VAX 8840, VAX DEC/Test Manager, and VMS.

[1] The VMM does not run on VAX 8820, 8830, or 8840 processors, due to microcode and console differences.

[2] At least one virtual machine must always run the VMS operating system, to carry out certain system management functions.

CH2884-5/90/0000/0002$01.00 © 1990 IEEE

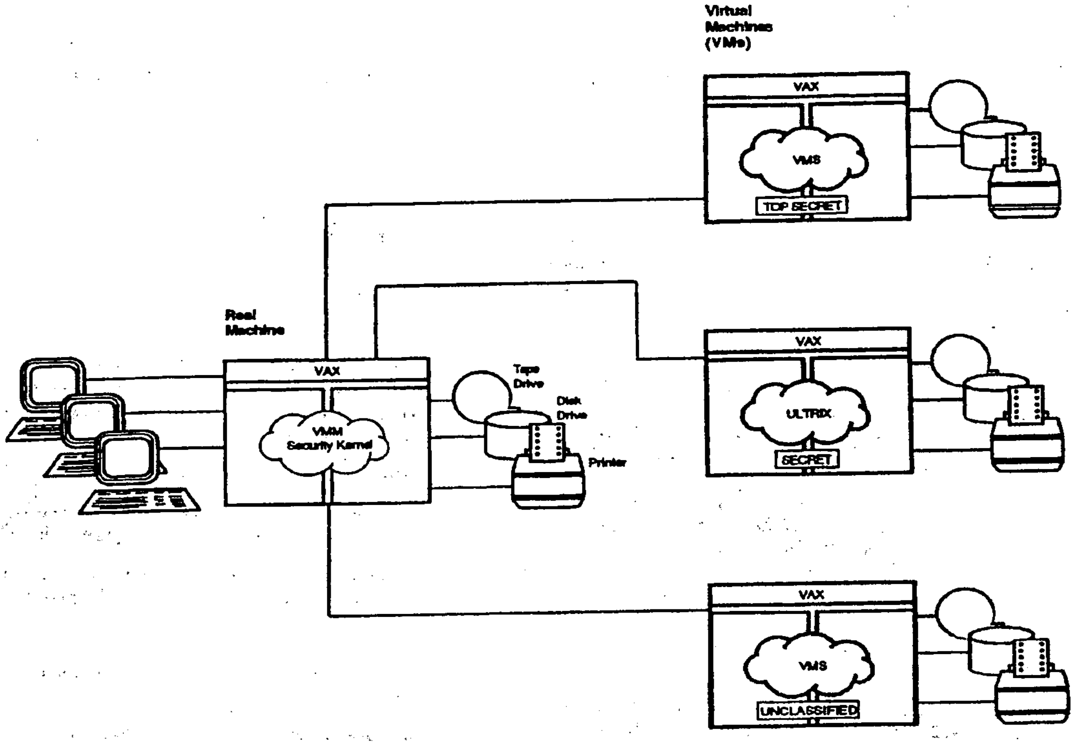

Figure 1: VAX VMM Security Kernel Configuration

The VAX security kernel applies both mandatory and discretionary access controls to virtual machines. Each virtual machine is assigned an *access class* that consists of a *secrecy class* and an *integrity class*, similar to those in the VMS Security Enhancement Service (VMS SES) [5]. The secrecy and integrity classes are based on the Bell and LaPadula security [2] and Biba integrity [4] models, respectively. The VAX security kernel also supports access control lists (ACLs) on all objects, similar to those in the VMS operating system [14].

The VMM security kernel is *not* a general purpose operating system. The principal subjects and objects are virtual machines and virtual disks, rather than conventional processes and files. That is the inherent difference between a VMM and a traditional operating system. Processes and files are implemented within the virtual machines by either the VMS or ULTRIX-32 operating systems.

The VAX security kernel can support large numbers of simultaneous users.[3] All software development of the VAX security kernel is now carried out on several virtual machines running on the VMM on a VAX 8800 system. On a typical day, about 40 software engineers and managers are logged in running a mixed load of text editing, compilation, system building, and document formatting. The system provides adequate interactive response time and is sufficiently reliable to support an engineering group that must meet strict milestones and schedules. As far as we know, the VAX security kernel is the first security kernel to support its own development team. The Multics Access Isolation Mechanism [36] was developed on Multics itself, but Multics with AIM was not a security kernel and only received a B2 rating.

The VAX security kernel is currently in the Design Analysis Phase with the National Computer Security Center (NCSC) for an A1 rating. It is being formally specified in Ina Jo and formal proofs are being done on the specifications.

3 Design Approach

This section describes several of the design choices in the VAX security kernel, including details about the virtual ma-

[3] Exact numbers depend on the precise hardware configuration.

chine approach to security kernels, virtualizing the VAX architecture, subjects and objects, access classes, our layered design, and other software engineering issues.

3.1 Virtual Machine Approach

The choice to build the VAX security kernel as a VMM was driven by two goals: to maintain compatibility with existing software written for the VAX architecture and to keep software development and maintenance costs to a minimum.

Digital Equipment Corporation began plans to enhance the security of the VAX architecture in mid-1979. Our initial effort was the design of security enhancements to the VMS operating system, first prototyped in 1980 and available today in the base VMS operating system and in the VMS Security Enhancement Service [5].

At the time of the initial prototype of the VMS security enhancements [16], Digital considered a traditional kernel/emulator security kernel to support VMS applications. However, it quickly became clear that the software development costs of a VMS emulator would be comparable to the cost of development of the VMS operating system itself. Worse still, the emulator would have to track all changes made to the VMS operating system, resulting in ongoing costs that would be unacceptably high for the limited market for A1-secure systems. The kernel/emulator system could not replace the existing VMS operating system because its performance would not be as good, and it would likely be export controlled. Furthermore, the growing demand for UNIX-based software would force development of a UNIX emulator at still more development cost.

To resolve these development cost and compatibility problems, we chose a VMM security kernel approach. A VMM security kernel presents the interface of a computer architecture that is comparatively simple and not subject to frequent change. Thus, the VAX security kernel presents an interface of the VAX architecture [21] and supports both the VMS and ULTRIX-32 operating systems with relatively few modifications.

The idea of a VMM security kernel is not a new one. Madnick and Donovan [22] first suggested the merits of VMMs for security, and Rhode [30] first proposed VMM security kernels. From 1976 to 1982, Systems Development Corporation (now a division of the UNISYS Corporation) built a kernelized version of IBM's VM/370 virtual-machine monitor, called KVM/370 [12]. While the design of the VAX security kernel is very different from KVM/370, we have applied some of the lessons learned in the KVM/370 project [11]. Section 7 compares the VAX security kernel with KVM/370. Gasser [10, Section 10.7] provides more detail on some of the trade-offs between a VMM security kernel approach and a kernel/emulator approach.

3.2 Virtualizing the VAX

The requirements for virtualizing a computer architecture were specified by Popek and Goldberg [26]. In essence, they require that all sensitive instructions and all references to sensitive data structures trap when executed by unprivileged code. A *sensitive* instruction or data structure is one that either reveals or modifies the privileged state of the processor.

3.2.1 Sensitive Instructions

Unfortunately, the VAX architecture does not meet Popek and Goldberg's requirements. Several instructions, including Move Processor Status Longword (MOVPSL), Probe (PROBEx), and Return from Exception or Interrupt (REI) are sensitive, but unprivileged. Furthermore, page table entries (PTEs) are sensitive data structures that can be read and written with unprivileged instructions.

As a result, we made a number of extensions to the VAX architecture to support virtualization. In particular, we added a VM bit to the processor status longword (PSL) that indicated whether or not the processor was executing in a virtual machine. A variety of sensitive instructions were changed to trap based on the setting of the VM bit, so that the VMM security kernel could emulate their execution. Space does not permit a full discussion of the instruction changes, but some details are discussed by Karger, Mason and Leonard [18].

3.2.2 Ring Compression

The most significant and security-relevant change to the VAX architecture was to virtualize protection rings. In the past, only processors with two protection states (such as the IBM 360/370 architecture) had been virtualized. Goldberg [13, section 4.3] described the difficulties of virtualizing machines with protection rings and therefore more than two protection states. He proposed several techniques for mapping ring numbers, some in software and one with a hardware ring relocation register, but he recognized that none of his techniques were satisfactory. His software techniques broke down because the physical ring number remained visible, and his hardware ring relocation technique broke down because virtualizing a machine with N rings always required N+1 rings.

Since the VMS operating system uses all four of the protection rings of the VAX architecture, it was essential that we develop a new technique for virtualization of protection rings. That technique is called *ring compression*.

Figure 2 shows how the protection rings of a virtual VAX processor are mapped to the rings of a real VAX processor. Virtual user and supervisor modes map to their real counterparts, but virtual executive and kernel modes both map to real executive mode. The real ring numbers are concealed from the virtual machine's operating system (VMOS) by three extensions to the VAX architecture: the addition of the VM bit to the PSL (described in Section 3.2.1), the addition of a VM processor status longword register (VMPSL), and the modification of all instructions that could reveal the real ring number. Those instructions either trap to the VMM security kernel for emulation or obtain their information from

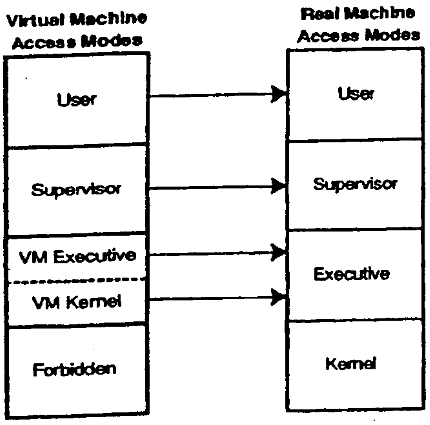

Figure 2: Ring Compression

the VMPSL, which contains the virtual ring number rather than the real ring number. Additional details can be found in Karger, Mason and Leonard [18].

Ring compression also requires that the security kernel change the memory protection of pages belonging to virtual machines so that their kernel-mode pages become accessible from executive mode. This change of memory protection could adversely affect security within a given virtual machine because the virtual machine's kernel mode is no longer fully protected from its executive mode.

For the two operating systems of interest to the VAX security kernel, there is no effective loss of security within the virtual machines themselves, although there is a loss of robustness against potentially bug-laden executive mode code. Fortunately, the VMS operating system grants all programs that run in executive mode the right to change mode to kernel at will and uses the the kernel/executive mode boundary only as a reliability mechanism. Furthermore, the ULTRIX-32 operating system does not use executive mode at all.

Of course, the compression of kernel and executive modes in the virtual machines in no way compromises the security of the VMM, as the security kernel runs only in real kernel mode, and no virtual machine ever is granted access to real kernel mode pages. If there were some other VAX operating system that actually used all four rings for security purposes, it would lose some of its own security, much as IBM operating systems lose some of their security when run in VM/370. However, no such operating systems exist for the VAX architecture.

3.2.3 I/O Emulation

Traditional virtual-machine monitors, such as IBM's VM/370, have virtualized not only the CPU, but also the I/O hardware. Virtualization of the I/O hardware allows the VMOS to run essentially unmodified. Virtualization of the VAX I/O hardware is particularly difficult because its I/O devices are programmed by reading and writing control and status registers (CSRs) that are located in a region of physical memory called I/O space. This type of I/O originated on the PDP-11 series of computers and caused performance difficulties in the UCLA PDP-11 virtual-machine monitor [27] because the VMM must somehow simulate every instruction that manipulates a CSR. Vahey [33] proposed a complex hardware performance assist, but such a device would add excessive complexity and development cost to the VAX security kernel.

Instead, the VAX security kernel implements a special I/O interconnection strategy for virtual machines. The VAX architecture [21] does not specify how I/O is to be done, and different VAX processors have implemented very different I/O interfaces. The VAX security kernel I/O interface is a specialized kernel call mechanism, optimized for performance, rather than traditional CSR-based I/O. In essence, a virtual machine stores I/O-related parameters (such as buffer addresses, etc.) in specified locations in its I/O space, but no I/O takes place until the virtual machine executes a Move to Privileged Register (MTPR) instruction to a special kernel call (KCALL) register. This MTPR traps to security kernel software that then performs the I/O. Thus, the number of traps to kernel software is dramatically less than would be required for CSR emulation.

This special kernel I/O interface means that special untrusted virtual device drivers had to be written for both the VMS and ULTRIX-32 operating systems, but this effort was no more than is typically required to support a new VAX processor, a small number of engineer-years.

Because the virtual VAX processors have an I/O interface different from that of any existing VAX processors, the VAX security kernel does not fall into any of Goldberg's traditional categories of VMMs. Goldberg [13, pp. 22-26] defines a Type I VMM as a VMM that runs on a bare machine. He defines a Type II VMM as a VMM that runs under an existing host operating system. Goldberg [13, section 3.3] also defines a Hybrid Virtual-Machine Monitor as one in which all supervisory-state instructions are simulated, rather than just the privileged instructions. The VMM security kernel is essentially a cross between a self-virtualizing Type I VMM for all non-I/O instructions and a Hybrid Virtual-Machine Monitor for I/O instructions.

3.2.4 Self-Virtualization

As we designed the extensions to the VAX architecture, we ensured that the architecture would permit *self-virtualization*. Self-virtualization is the ability of a virtual-machine monitor to run in one of its own virtual machines and recursively create second-level virtual machines. Self-virtualization is very useful for developing and debugging the virtual-machine monitor itself, but it is of little value to actual users. Since self-virtualization would have added significant complexity to the Trusted Computing Base (TCB), no software support has been done.[4]

3.3 Subjects

There are two kinds of subjects in the VAX security kernel, users and virtual machines (VMs). A user communicates over the trusted path with a process called a *Server*. Servers are trusted processes, but unlike the trusted processes in other systems such as KSOS-11 [3], servers run only within the kernel itself. User subjects cannot run user-written code; servers execute only verified code that is part of the TCB.

The powers of a server are determined by:

- The user's minimum and maximum access class. (See Section 3.5.)
- The terminal's minimum and maximum access class.
- The user's discretionary access rights.
- The user's privileges. (See Section 3.6.)
- The privileges exercisable from the terminal.

A virtual machine is an untrusted subject that runs a VMOS. A user interacts with the VMOS in whatever fashion is normal for that operating system, for example, by logging into that VMOS and issuing commands. A user may write and run code inside a VM and even penetrate the VMOS, all without affecting the security of other virtual machines or the security kernel itself. At worst, a penetrated virtual machine could only affect other virtual machines with which it shared disk volumes.

On login to the security kernel, the VMM establishes a connection between the user's terminal line and the user's Server, called a *session*. When the user wants to use some virtual machine, the user issues the CONNECT command to his or her Server, specifying the name of that VM. If the connection is authorized, the system suspends the user's existing session with the Server and establishes a new session between the user's terminal line and the requested virtual machine. Thus, the Servers and the VMs have distinct identities and distinct security attributes.

Virtual machines may be run in a single-user mode to provide maximum individual accountability. Alternately, they can be run in a multj-user mode. In such a case, individual accountability might be achieved by running a VMOS with at least a C2 rating, as suggested by the proposed Trusted VMM Interpretation [1] of Trusted Information Systems, Inc.

Virtual machines can also be treated as objects because a user may request that the TCB provide a connection between the user's terminal and some VM. For this operation, the user is the subject and the VM is the object.

3.4 Objects

The VAX security kernel supports a variety of objects including real devices and volumes and security kernel files.

One group of objects comprises the real devices on the system: disk drives, tape drives, printers, terminal lines, and single access-class network lines. As these devices can contain or transmit information, access to them must be controlled by the TCB. Another object is the primary memory that is allocated to each VM when it is activated.

Disk and tape volumes are also objects. The contents of some disk volumes are completely under the control of a virtual machine. They may contain a file system structure or just an arbitrary collection of bits, depending on the method used by the VMOS to access the volume. Such volumes are called *exchangeable volumes* because they may be exchanged with other computer systems running conventional operating systems. Other disk volumes contain a VAX security kernel file structure and are called *VAX security kernel volumes*. These volumes must not be directly accessed by a VMOS or exchanged with other systems, as an untrusted subject could subvert the kernel's file system or read information to which it was not entitled. The VAX security kernel does not provide trusted tape volumes; all tape volumes are exchangeable.

VAX security kernel volumes contain VAX security kernel files that are organized as a flat file system. VAX security kernel files are used for a variety of purposes in the system and are considered objects by the TCB. One use for VAX security kernel files is to hold long-term system databases such as the audit log and the authorization file. These files are considered part of the TCB and, with the exception of the audit log, error log and crash dump files, cannot be directly referenced by virtual machines.

Another use of VAX security kernel files is to create virtual disk volumes, loosely analogous to mini-disks in IBM's VM/370 [23, pp. 549-563]. Mini-disks allow a physical disk to be partitioned, so that one need not dedicate an entire physical disk to a small virtual machine that only requires a small amount of disk space. Such virtual disks may contain the file structure of some VMOS, such as a VMS file structure or an ULTRIX 32 file structure. However, the VMM deals with virtual disks only as a whole. The contents of a virtual disk are all part of a single object as far as the VMM is concerned.

3.5 Access Classes

The VAX security kernel enforces mandatory controls, as required of all A1 systems. Both secrecy and integrity models

[4] The software changes needed for self-virtualization primarily consist of changes to the virtual device drivers described in Section 3.2.3 and some changes in the emulation of certain sensitive instructions. Under the proposed Trusted VMM Interpretation [1], it might even be possible for a self-virtualized security kernel to itself remain A1 rated. To achieve that goal, the first level VMM would map the second level VMM's kernel mode to real executive mode, while the VMs running on top of the second level VMM would have their supervisor, executive, and kernel modes all mapped to real supervisor mode. Of course, as one continues to recursively self-virtualize, one runs out of protection rings at the fourth level VMM, and that VMM would no longer be protected from its virtual machines.

are supported, based on the work of Bell and LaPadula [2] and of Biba [4], respectively. To implement mandatory controls, each kernel subject and kernel object is assigned a sensitivity label, called an *access class*.[5] An access class consists of two components, a *secrecy class* and an *integrity class*. These components are each further divided into a level and a category set. A *secrecy level* is a hierarchical classification. The *secrecy category set* is the set of non-hierarchical secrecy categories that represents the sensitivity of the access class. The integrity level and integrity category set are defined analogously. For compatibility with VMS SES [5], the kernel supports 256 secrecy levels, 256 integrity levels, 64 secrecy categories, and 64 integrity categories.

Given the complex structure of access classes, two definitions must be carefully constructed:

Definition 1 *An access class A is* equal *to an access class B if and only if:*

- *The secrecy level of A is equal to the secrecy level of B,*
- *The secrecy category set of A is equal to the secrecy category set of B,*
- *The integrity level of A is equal to the integrity level of B, and*
- *The integrity category set of A is equal to the integrity category set of B.*

Definition 2 *An access class A* dominates *an access class B if and only if:*

- *The secrecy level of A is greater than or equal to the secrecy level of B,*
- *The secrecy category set of A is an improper superset of the secrecy category set of B,*
- *The integrity level of A is less than or equal to the integrity level of B, and*
- *The integrity category set of A is an improper subset of the integrity category set of B.*

The secrecy and integrity models define that a subject may reference an object depending on the access classes of the subject and object and on the type of reference. A subject may read from an object only if the subject's access class dominates the access class of the object. A subject may write to an object only if the object's access class dominates the access class of the subject.[6] Thus, for example, a virtual machine may mount for read-write access an exchangeable volume only if the VM's access class is equal to that of the volume. However, the VM may mount for read-only access any exchangeable volume where the VM's access class dominates that of the volume.

3.6 Privileges

System managers, security managers, and operators gain their powers by having *privileges*. The privileges allow great flexibility in the assignment of powers and responsibilities, including a measure of two-person control and separation of duties. Privileges restrict access beyond the protections provided by mandatory and discretionary access controls. A privileged user cannot see data that would be otherwise inaccessible. Only the downgrading privileges allow bypassing of access controls, and the use of those privileges is audited.

Most privileges can be exercised only through the trusted path and are called *user privileges*. (See Table 1.) Two privileges can be exercised by virtual machines and are called *virtual-machine privileges*. (See Table 2.)

3.7 Layered Design

The VAX security kernel has been implemented following the strict *levels of abstraction* approach originally used by Dijkstra [8] in the THE system. Janson [15] developed the use of levels of abstraction in security kernel design as a means of reducing complexity and providing precise and understandable specifications. Each layer of the design implements some abstraction in part by making calls on lower layers. In no case does a lower layer invoke or depend upon higher layer abstractions. By making lower layers unaware of higher abstractions, we reduce the total number of interactions in the system and thereby reduce the overall complexity. Furthermore, each layer can be tested in isolation from all higher layers, allowing debugging to proceed in an orderly fashion, rather than haphazardly throughout the system. This type of layering is called out in the requirements for B3 and A1 systems when the NCSC evaluation criteria [7, p. 38] state that, "The TCB shall incorporate significant use of layering, abstraction and data hiding. Significant system engineering shall be directed toward minimizing the complexity of the TCB ..."

The layered design of the VAX security kernel was based heavily on the Multics kernel design work of Janson [15] and Reed [28] and to a lesser extent on the Naval Postgraduate School kernel design [6]. Figure 3 shows a diagram of the layers of the VAX security kernel. The arrows in the diagram indicate how each layer functionally depends on the abstract machine created by lower layers.

Each layer adds specific functions within the security kernel, such that at the security perimeter, the secrecy and integrity models are enforced. The kernel itself is process structured, as described in the summary of the various layers. Unlike many other kernels, all of the trusted processes run within the security perimeter and are included in the formal specifications described in Section 5.4.

HIH The Hardware-Interrupt Handler layer is immediately above the physical VAX hardware and modified microcode. It contains the interrupt handlers for the various I/O controllers and certain CPU-specific code.

[5] Some objects, such as terminal lines, may be assigned a range of access classes.

[6] In general, write access is even further restricted; a subject may write to an object only if the subject's and object's access classes are equal. This disallows blind writes to an object that cannot be read.

Privilege	Powers
CLASSIFY_DEVICE	Assign access classes to I/O devices and privileges to terminals
CLASSIFY_SUBJECT	Assign access classes and privileges to subjects; name levels and categories
CLASSIFY_VOLUME	Register and assign access classes to volumes
DELETE_AUDIT	Delete audit data
DOWNGRADE_SECRECY	Downgrade secrecy of text after human inspection
DOWNGRADE_SECRECY_NOINSPECT	Downgrade secrecy of data without inspection
ENABLE_DEBUGGER	Enable untrusted kernel debugger
OPERATE	Mount volumes, change printer paper, boot and shutdown system
REGISTER	Register and change non-security attributes of devices, virtual machines, and users
SET_AUDIT	Control audit log and real-time alarms
SET_COVERT_CHANNEL_DEFENSE	Enable or disable covert channel defenses
SET_FILE	Create, delete, or copy kernel files
SET_PASSWORD	Change users' passwords and password parameters
UPGRADE_INTEGRITY	Upgrade integrity of text after human inspection
UPGRADE_INTEGRITY_NOINSPECT	Upgrade integrity of data without inspection

Table 1: User Privileges

Privilege	Powers
OPERATE	Dismount volumes; activate and deactivate other virtual machines; set login limits
SET_ACL	Change any object's ACL, if access class permits

Table 2: Virtual Machine Privileges

LLS The Lower-Level Scheduler is based strongly on Reed's two-level scheduler design [28]. It creates the abstractions of level one virtual processors (vp1s) that are the basic unit of scheduling for the system. The LLS supports symmetric multiprocessing by binding and unbinding real CPUs to individual vp1s. As shown in Figure 4, there are three kinds of vp1s: *dedicated* vp1s that typically contain device drivers, *bindable* vp1s that can be bound to dedicated vp2s by the higher level scheduler, and *addressable* vp1s that can be bound to bindable vp2s and thereby to virtual machines. Vp1s are intended to be very inexpensive processes for use within the kernel. Only addressable vp1s have full address spaces; all other vp1s run out of the global address space of the kernel. Thus, the lower-level scheduler can context switch in and out of most vp1s by merely saving registers and switching stack pointers. The lower-level scheduler also implements eventcounts [29] as the basic synchronization mechanism of the kernel. Eventcounts can be awaited or advanced in the normal way, or a *processor interrupt* can be tied to an eventcount, such that a VM can be interrupted when an eventcount has reached a particular value. This processor interrupt mechanism provides upward transfers of control that are otherwise forbidden in the kernel. Processor interrupts are only delivered when the CPU is executing outside the security kernel.

IOS The I/O services layer implements device drivers that control the real I/O devices. The current version supports only directly connected terminals and storage devices.

VMP The VM physical memory layer manages real physical memory, and assigns it to virtual machines.

VMV The VM virtual memory layer implements the *shadow page tables* needed to support virtual memory in the virtual machines.[7] VMV implements a primary-memory only strategy, requiring that all the physical memory that a virtual machine sees be physically resident when that virtual machine is active. While this technique limits the number of simultaneously active

[7] Shadow page tables are created by a VMM, because the physical addresses in page table entries must be relocated. Shadow page tables are described in detail by Madnick and Donovan [23, Section 9-5]. Shadow page tables are also where ring compression occurs.

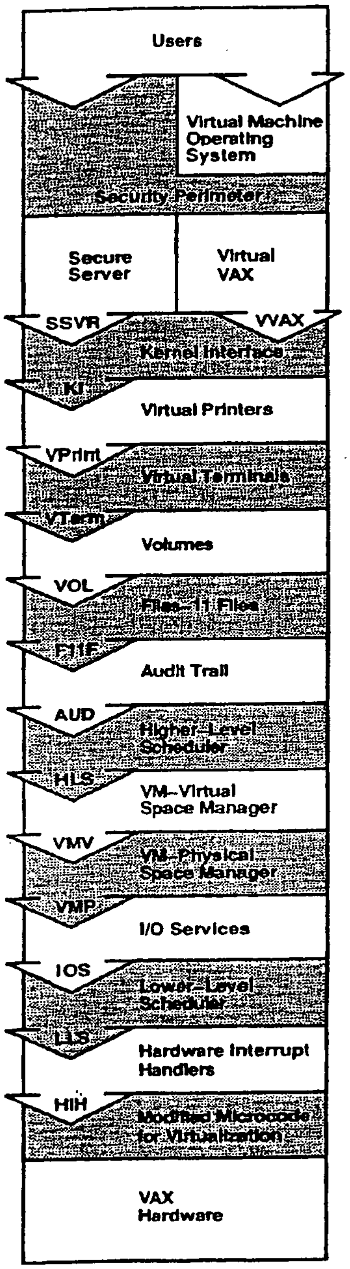

Figure 3: VAX Security Kernel Layers

virtual machines to the number that can fit into physical memory simultaneously, it significantly reduces kernel complexity by eliminating the need for a demand-paging mechanism in the kernel. It also eliminates the phenomenon of *double paging* that is often seen in other VMMs, where the demand paging mechanisms of the VMM and of the VMOS can thrash against one another, leading to serious performance degradation. In the VMM security kernel, the virtual machines are allocated a fixed amount of physical memory and do all their own paging.

HLS The Higher-Level Scheduler is also based on Reed's two-level scheduler [28]. Unlike Reed's design, our higher-level scheduler is extremely simple because it does not need to schedule access to primary memory. The HLS does create the abstraction of level-two virtual processors (vp2s). There are two kinds of vp2s: *dedicated* vp2s that are used primarily by the SSVR layer described below and *bindable* vp2s that are used for virtual machines. Figure 4 shows the relationships between vp1s and vp2s.

AUD The auditing layer provides the facilities for security auditing and security alarms. It is described in detail in a companion paper [31].

F11F The Files 11 Files layer implements a subset of the ODS 2 file system that is also used in the VMS operating system.[8] The most significant restrictions on the VAX security kernel implementation of ODS 2 are that all files must be pre-allocated and contiguous. This reduces kernel complexity by eliminating the need for dynamic file extensions. F11F implements ODS 2 files only as a flat file system.

VOL The Volumes layer implements VAX security kernel and exchangeable volumes and provides registries of all subjects and objects. These registries are much simpler than ODS 2 directories.

VTerm The Virtual Terminals layer implements virtual terminals for each virtual machine, and manages the physical terminal lines. Each user may have multiple sessions connected to different virtual machines, and VTerm provides the session management functions, as described in Section 4.1. VTerm also implements asynchronous network lines to allow virtual machines to connect to single-access-class networks via specially dedicated terminal lines. The current version of the system has no support for higher-speed network connections.

VPrint The Virtual Printers layer implements virtual printers for each virtual machine and multiplexes the real physical printers among the virtual printers. It provides top and bottom labeling, as well as trusted banner pages to delimit listings of different access classes and different VMs.

[8] A brief summary of the Files-11 ODS 2 structure can be found in the appendices of [35].

KI The Kernel Interface layer implements virtual controllers for the various virtual I/O devices and the security function controller, which implements such functions as loading virtual disks into virtual drives.

VVAX The Virtual VAX layer completes the virtualization process by emulating sensitive instructions, delivering exceptions and interrupts to the virtual machine, etc.

SSVR The Secure Server layer implements the trusted path for the security kernel, logs users in and out, and provides security-related administrative functions. There is a dedicated vp2 for each terminal line to provide a Server process for each logged in user.

VMOS The VMOS layer is the virtual machine's operating system.

USERS The users in Figure 3 include both the untrusted applications programs that run on top of the VMOS, and the human beings who communicate directly with the secure server via the trusted path.

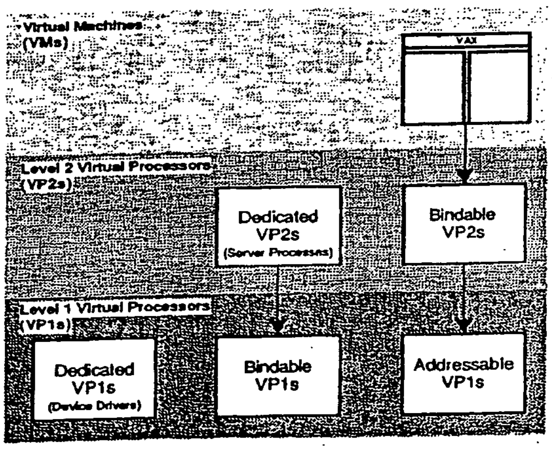
Virtual Machines (VMs)
VAX
Level 2 Virtual Processors (VP2s)
Dedicated VP2s (Server Processes)
Bindable VP2s
Level 1 Virtual Processors (VP1s)
Dedicated VP1s (Device Drivers)
Bindable VP1s
Addressable VP1s

Figure 4: Level One and Level Two Virtual Processors

3.8 Software Engineering Issues

A number of interesting software engineering issues arose during the development of the VAX security kernel. While space does not permit discussing all of them, this section highlights a few of the most significant.

3.8.1 Programming Language Choice

Perhaps the most critical software engineering issue in the VAX security kernel design was the choice of a programming language. From the problems that KSOS 11 had with its choice of compilers [3, 25], it was clear that we needed high quality compilers to develop our security kernel. While we wanted as strongly-typed a language as possible, it was much more critical that the compiler correctly compile very large programs, produce high quality VAX object code, and be supported by an organization that could quickly respond to any problems we might find.

At the time the VAX security kernel prototype effort began, there were only a small number of systems programming languages available for the VAX architecture: BLISS 32, PL/I, PASCAL, and C. BLISS 32 was rejected because of its lack of data typing facilities. PASCAL was rejected because the V2.0 compiler that generated high quality code was not yet available. This left PL/I and C, both of which used the same good quality code generator. We chose PL/I because of its slightly better data typing support, because of its better support for character string manipulation, and because the first prototype developers had extensive prior experience in coding operating systems in PL/I.

We were not happy with the choice of PL/I because its data types were not strongly enforced. When the high quality V2.0 PASCAL compiler became available, we began writing new code for the kernel in PASCAL. PASCAL provides much stronger data-type checking than PL/I, and the VAX calling standard made inter-language calls easy to implement.

Higher-level language compilers cannot generate optimal code for all programs. Therefore, we found it necessary to implement those modules that actual measurements had shown to be performance-critical in the MACRO 32 assembly language. Table 3 shows how much code was written in each of the languages for each layer of the kernel.[9] The table shows the number of executable source code statements (excluding comments, declarations, and white space) and per-layer and per-language totals.

In retrospect, the use of both PL/I and PASCAL has led to certain difficulties. Software engineers must be trained in both languages, and some kernel bugs have resulted from misunderstandings of how to pass parameters from one language to the other. Future security kernel developers would do well to choose one systems programming language and stick to it.

3.8.2 Coding Strategies

A number of coding strategies proved very useful in the development of the VAX security kernel. For example, we avoided all use of global pools within the kernel to minimize the possibility of storage channels. The maximum size of data structures is determined at system boot time (based

[9] Table 3 includes a number of entries that are not shown in the layer diagram in Figure 3. These layers, COMMON, PMM, SVSBOO, VMMBOOT, and VMMLIB provide certain booting and runtime library support functions. The normal runtime libraries for the PL/I and PASCAL languages are not linked into the kernel because they would have added a large amount of code that would need to be evaluated and placed under configuration control.

Layer	MACRO	PASCAL	PL/I	Total
VVAX	3371	1502	0	4873
SSVR	0	6876	330	7206
KI	10	3354	0	3364
VPRINT	0	1455	0	1455
VTERM	0	1419	0	1419
VOL	0	2553	0	2553
F11F	0	2962	0	2962
AUD	0	543	0	543
HLS	0	0	430	430
VMV	129	0	1069	1198
VMP	0	0	352	352
IOS	0	4725	0	4725
LLS	1289	13	3839	5141
HIH	815	2393	174	3382
COMMON	244	0	0	244
PMM	0	0	176	176
SVSR00	2541	734	0	3275
VMMBOOT	55	213	430	698
VMMLIB	3021	503	1265	4789
Total	11475	29245	8065	48785

Table 3: Executable Statements per Layer

on system generation parameters), and memory is allocated for that maximum size during kernel initialization.

Different sections of memory within the kernel are separated by no-access guard pages to detect run-away array or string references. Unused memory is set to all ones to increase the chance of detecting the use of uninitialized variables because zeros are less likely to generate exceptions.

The layers of the kernel are coded defensively with sanity checks to protect each layer from higher layers. If irregularities are detected, the system crashes to avoid the possibility of a security compromise. These sanity checks were devised to aid in the debugging of the kernel and do not themselves provide security assurance mechanisms. However, many of the checks remain enabled in the finished kernel to help detect any remaining bugs.

The actions of a user or a virtual machine cannot crash the kernel. They can cause error messages, exception conditions raised in the virtual machine, or in extreme cases, the halting of an offending subject.

Since the entire TCB runs in kernel mode, there are no hardware-enforced firewalls between layers. However, the layering methodology forbids lower layers from calling higher layers. To help us spot layer violations, we applied both automatic and manual techniques. Using the features of the VAX DEC/Module Management System (VAX DEC/MMS) and the VAX DEC/Code Management Systems (VAX DEC/CMS), we were able to isolate all dependencies of a layer on other layers. By visual inspection, we could immediately spot upward references. In fact during development, we did detect and fix several such occurrences.

4 Human Interfaces

High-security systems have developed a reputation for being hard to use, primarily due to their limited user interfaces. We believe that it is essential that a human interface meet the expectations of today's commercial computer users. However, we faced the same obstacles faced by other developers of high-security systems:

- Development resources are limited and satisfying the A1 criteria takes precedence over all other efforts.
- The kernel must be small and verifiable. User interface features, such as a sophisticated command parser, are large and often difficult to verify. Consequently, an interface built entirely on trusted code cannot match the usability of an interface built on untrusted code.

We overcame these obstacles by creating two separate command sets: the Secure Server commands and the SECURE commands. The Secure Server commands are implemented entirely in trusted code. The administrative commands, the SECURE commands, are parsed in the VMS and ULTRIX 32 operating systems. With this approach, we reduce the amount of trusted code and gain the well-developed command interfaces of these mature commercial operating systems. SECURE commands are normally only issued by the system manager, the security manager, the operators, and the auditors, although ordinary users may need to issue a few of them at times. By contrast, all users must issue some Secure Server commands to login and connect to virtual machines.

4.1 Secure Server Commands

The Secure Server is the user's direct interface to the kernel. A user invokes a trusted path to the Secure Server by pressing the *Secure Attention Key*. This key operates at all times and cannot be intercepted by untrusted code. We have chosen the BREAK key to be the Secure Attention Key.

The Secure Server's commands control terminal connections to virtual machines in the same way that a terminal server controls terminal connections to physical machines, using commands such as: **CONNECT**, **DISCONNECT**, **RESUME**, and **SHOW SESSIONS**. A user can create sessions with several virtual machines at different access classes and can quickly switch from one to another.

The interface for the Secure Server commands is built entirely in trusted code and offers only minimal command-line editing functions.

4.2 SECURE Commands

The tools for managing the system are the SECURE commands. The SECURE commands and utilities are implemented just as are other commands in the VMS and ULTRIX-32 command languages, except that they issue kernel calls to do their work. The complete set of SECURE commands and utilities is installed in the VMS operating system. A subset of the SECURE commands is offered by the ULTRIX-32 operating system.

The SECURE commands, unlike the Secure Server commands, are parsed by the VMS and ULTRIX 32 command language interpreters. The user can take advantage of such features as command-line recall and command procedures.

There are two types of SECURE commands: *VM SECURE* commands and *User SECURE* commands. Both types of SECURE commands are issued from the VM's operating-system command level. VM SECURE commands are executed in the context of the issuing VM. User SECURE commands are submitted to the Secure Server for execution. The commands are distinguished by the type of subject, a user or a virtual machine, that holds the access class and privileges necessary to issue the command.

4.3 Command Confirmation

While both the User and VM SECURE commands are administrative commands, only the User SECURE commands must be trusted. For such security-relevant commands, we require A1 assurance that:

- The command was issued by a user and not by a Trojan horse in a VM.
- The command received by the Secure Server is exactly the same command typed by the user and not a command that was covertly modified by a Trojan horse.
- The user who issued the command can be identified in the audit log.

Our design for the User SECURE commands provides both trust and individuality accountability even for commands issued from an untrusted environment. Upon receipt of a valid User SECURE command, the VM instructs the user to press SECURE ATTENTION. This key invokes a trusted path between the user's terminal and the Secure Server. A SECURE ATTENTION signal can be sent to the Secure Server only by manually pressing the BREAK key. This prevents a Trojan horse from completing the execution of a User SECURE command.

To prevent a VM from spoofing the user by passing a different command from what the user typed, the Secure Server displays the action that will be taken by the command and prompts the user to approve or reject the operation. Figure 5 is an abbreviated example of a User SECURE command issued from a VMS virtual machine. *Resuming* indicates that control of the terminal will be returned to the virtual machine.

4.4 SECURE Utilities

Managing the VMM security kernel requires a number of utilities. Our SECURE utilities are modeled after VMS utilities and are summarized in Table 4.

```
$ SECURE DELETE TLS:STATUS.RPT
Press SECURE ATTENTION to complete
   execution of this command.
User presses SECURE ATTENTION to establish a
   trusted path.
Delete VAX security kernel file
   TLS:STATUS.RPT
Confirmation [Yes or No]:  Y
VMM: File deleted
Resuming...
```

Figure 5: Example of a User SECURE command

SECURE Utility	Purpose
Authorize	Registers users and virtual machines, etc.
Register/Device	Registers I/O devices.
Register/Volume	Registers disk and tape volumes.
Sysgen	Sets limits on system resources.
Crash Dump Analyzer	Provides data for determining the cause of a system crash.

Table 4: SECURE Utilities

4.5 Reclassifying Information

Users can be permitted to change the access class of the contents of a VAX security kernel file or an exchangeable volume with the SECURE RECLASSIFY command. This command copies the contents of a kernel file or volume to an existing kernel file or volume labeled with a different access class. The source and destination objects must lie within the user's access-class range. In addition, privileges are required if the reclassification downgrades the data's secrecy class or upgrades its integrity class.

Reclassification normally requires trusted inspection by the user. Inspection is required to be sure that a Trojan horse has not inserted additional information that the user did not intend to reclassify. To make inspection easier, the user can opt to print the VAX security kernel file or display the file on the terminal, one screen at a time. Once the complete file is printed or displayed, the user is prompted to approve the reclassification. To prevent the covert passing of information from the source file to the target file in the form of invisible escape sequences, inspected files must contain only printing characters, spaces, and form feeds. A line may not end with a space because a trailing space would be invisible. The reclassification is terminated if any illegal character is encountered.

5 Assurance

The principal reason for building an A1 security kernel is to provide a high degree of assurance that the security features of the system actually work correctly. This section describes some of the techniques that we have used in the VAX security kernel to provide the necessary assurance of security, to meet both the requirements of an A1 evaluation and the requirements of real-world users. It is this integration of both A1 requirements and real-world requirements that is of particular research interest, as previous security kernels have not succeeded at integrating the A1 requirements with good performance and compatibility with large amounts of existing commercial software.

Gasser [10, p. 163] describes Honeywell's STOP kernel for the SCOMP [9] and Gemini Computers' GEMSOS [32] as commercial-grade security kernels. However, STOP does not provide software compatibility with existing operating systems, and GEMSOS to date has only been used in specialized environments. Shockley, Tao, and Thompson [32] report that research is under way to provide both UNIX and MS-DOS environments for GEMSOS, but it is not clear whether those environments are yet working. If Gemini succeeds in providing both UNIX and MS-DOS environments in GEMSOS, they will have succeeded at integrating A1 requirements with real-world requirements. The VAX security kernel supports both the VMS and ULTRIX-32 operating systems with their layered applications today.

5.1 Design and Code Changes

Every change to our code undergoes both design and code review, regardless of whether the code is trusted or untrusted, or whether it is a whole new layer or a bug fix. Design reviews for even the smallest fixes ensure that system-wide effects are considered. Each layer has an owner, who participates in the design review, and is responsible for the quality of that layer. Each code change is reviewed both in the context of its own layer and in the contexts of its calling and called layers, so as to catch inter-layer problems.

Reviewers learn from the code they review, as well as sharing their knowledge through review comments. Reviewers address readability and clarity, security, performance, elegance and adherence to guidelines. Much like access controls, design and code guidelines are either mandatory or discretionary. Mandatory guidelines are based on prior experience in security kernel developments. Discretionary guidelines are used to avoid well-known traps in the programming language, and to produce consistent, readable code. This consistency makes it easier for an engineer to pick up and debug in a new area, reducing engineering costs and time.

The code review results, along with the design and test plan, are publicized for the entire group to check. This practice provides a last review of the entire change by a large audience. Code review results can also serve as examples from which engineers can learn good coding practices.

The development team makes extensive use of VAX Notes online conferences to publicize design and coding guidelines, to discuss specific design issues, to track bug reports, and to record and publicize the results of the above-mentioned design and code reviews.

Each coding task is integrated with the current working system as soon as it is complete. This integration always produces a working system. (See Section 5.3.) Continual and incremental integration avoids major unexpected failures by identifying design and/or coding errors as soon as possible.

5.2 Development Environment

As mentioned in Section 2, we have been developing the VAX security kernel on a VAX security kernel system. Thus, our group does its daily work on a system designed to meet A1 security requirements, using most of its features and controls. Our VMs run at meaningful access classes. Different versions of the kernel are maintained on different VMs to keep orthogonal tasks from impinging on each other. We also use VMs for developing and testing the untrusted code that must run in the VMS and ULTRIX-32 operating systems. We have separated the roles of our own system manager and security manager, as recommended in the NCSC Evaluation Criteria [7].

The CPU and console of the development machine are kept inside a lab that only members of the VAX security kernel development group can enter. Within that lab, the development machine is protected by a *cage*, which consists of another room with a locked door. Physical access to both the lab and to the cage within the lab is controlled by a key-card security system. Finally, our development machine is not yet connected to Digital's internal computer network, to minimize the external threat to our development environment and our project.

5.3 Testing

Integrating a coding task requires that a developer run a standard regression test suite. Integration occurs usually at least once a week, and as often as twice a day.[10] This regression suite consists of two portions: *layer tests* and *KCALL tests*. Layer tests are linked directly into the kernel, and test layer interfaces and internal routines by calling them directly and checking their outcome. KCALL tests run in a VM, issuing legal, illegal, and malformed requests, to check the VM interface.

A separate suite of tests, issued via the VAX DEC/Test Manager (DTM), is run once every two weeks to test the user command interface. These tests currently run for 30 hours. They consist of commands that are successful, commands that produce errors, and commands that send malformed packets to the SSVR layer. DTM checks both the results of each command and the displays it produces.

We also run the standard VAX architecture exerciser (AXE) that verifies that a particular CPU correctly imple-

[10] Developers of course run individual tests prior to integration.

ments a VAX computer. We run AXE to test the VAX virtualization, described in Section 3.2. AXE tests were run extensively during the development of the CPU microcode extensions and the VVAX layer. They will be run again when the kernel reaches final completion.

We are currently developing test plans for fully exercising all of the access control decisions and other security-relevant checks made by the system and for system-penetration testing. Some of these new tests will be developed from scratch, and some will be based on the formal specifications.

5.4 Formal Methods

The requirements for an A1 security evaluation state that a formal security policy model must be written, that a formal top-level specification (FTLS) of the system design must be written and proven to satisfy the security policy model, that the system implementation must be informally shown to be consistent with the FTLS, and that formal methods must be used in covert channel analysis of the system. The FTLS must accurately model system external interfaces, externally visible behavior, and security-relevant actions. A descriptive top-level specification (DTLS) is also required as a complete natural language description of the system.

We use the Formal Development Methodology (FDM) specification and verification system [19] to help meet these requirements. We are writing both our security policy model (which consists of criteria and constraints and the top-level specification (TLS) of the various transforms) and our FTLS in the FDM specification language, Ina Jo. We are using the FDM interactive theorem prover (ITP) to show that the TLS obeys the policy and that the FTLS maps to the TLS. The DTLS consists of our internal design documentation, plus some special *glue documents* that tie the DTLS and the FTLS together, particularly describing areas of the kernel that are not formally modeled in the FTLS.

Table 3 shows the number of executable statements in the security kernel. For comparison, table 5 shows an estimate of the total number of lines of Ina Jo (comments excluded) and the number of lines of transforms (declarations excluded) required to specify that kernel. The numbers are estimates because the FTLS is not yet complete. The totals show that the number of lines of transforms are about one sixth of the number of executable statements in the security kernel.

Level of Specification	Lines of Ina Jo	
	Total	Transforms
TLS	650	294
FTLS	11758	8410
Total	12408	8704

Table 5: Lines of Formal Specifications

We are doing a formal covert channel analysis using a new technique for automating the Shared-Resource Matrix approach [20] using code-level flow analysis tools.

Formal methods do not make the system secure by themselves. Successful proof that our specifications meet security policy does not guarantee that there are no lurking implementation bugs. However, the use of formal methods significantly improves the overall quality of the security kernel. When combined with the informal testing procedures of Section 5.3, the use of formal methods improves the assurance that the security features are effective. Indeed, the very act of formally specifying the security kernel in Ina Jo has already detected several kernel bugs, both because of constraints imposed by proof procedures, and because the process of code correspondence provides a thorough method for reviewing the TCB code and informal design specifications. The separation of duties between the software engineer and the verifier, by itself, provides valuable extra assurance, even if no proofs were ever done.

5.5 Configuration Control

We maintain strict configuration control on many items, including design documents, trusted kernel code, test suites, user documents, and verification documents. All of our code is maintained under the VAX DEC/Code Management System (CMS) to maintain a history of each change to each module. Security reviews check each item against the specific NCSC criteria requirements [7] it fulfills and check among the items for internal consistency. Items that have been reviewed are stored on a master pack that is physically protected against modification.

Our hardware, firmware, and software development tools are developed by other groups within the corporation. We review hardware and firmware ECOs, prior to supporting them in the VAX security kernel. New versions of software development tools are tested on a stand-alone laboratory system prior to use on the kernel development machine. We use only the standard, released versions of software development tools, the same versions that have been checked out for shipment to our customers. With rare exceptions, no field-test versions are permitted on the kernel development machine.

5.6 Trusted Distribution

The end user of a security kernel must have some assurance that no one has tampered with or substituted counterfeit copies of the hardware and software that make up the system. Hardware and software have different trusted distribution requirements.

5.6.1 Hardware Trusted Distribution

To assure that the hardware systems would arrive at the customer's site meeting the trusted distribution criteria, we have developed a security-seal program. If someone tampered with the seal, evidence would be provided of the attempted entry. A locking device would combine with the security sealing procedures to ensure a trusted shipment. Full individual accountability would be provided, including logs of the delivery.

5.6.2 Software Trusted Distribution

Installation of an A1 system involves achieving a trusted state. The steps to do this on VAX 8800 hardware are complex. The console processor software and CPU microcode must be installed and cryptographically checksummed with stand-alone software to detect any possible tampering. If a secure site loses its trusted state for any reason, they must reinstall the console software and the CPU microcode. Trusted state could be lost just by running an untrusted operating system or hardware diagnostics on the system.

Next, the trusted code is installed via untrusted code (VMS) and the result is cryptographically checksummed to verify that the untrusted code has not tampered with the trusted code. The result of the checksum is checked against a *message authentication code* to verify correct installation. The checksumming software is shipped separately from the rest of the software, so that a single failure of the trusted distribution system could not compromise both the checksum program and the authentication code.

For software, there would also be an option of using trusted couriers instead of the separate delivery paths.

6 Production-Quality Kernels

A production-quality security kernel is designed to protect and ensure the quality of real-world information. This section describes some of the differences between research and production-quality security kernels that are required to meet general user requirements, as well as to satisfy the NCSC criteria for an A1 operating system.

6.1 Producing the Kernel

The primary tools for creating a security kernel are compilers. Quality compilers must work for large programs, produce efficient object code, and be reliably supported. We sacrificed programming language elegance in favor of compilers with a strong track record: the VAX PASCAL and PL/I compilers. We maintained contact with the compiler developers throughout the development, and they provided much needed help to us, including occasional changes to the actual compiler code.

A second tool, a symbolic debugger/crash dump analyzer, is needed to develop and debug the system. It would also be needed by users and support personnel to diagnose problems, and by users who might wish to add functions to the kernel.

A production-quality security kernel must have adequate performance to justify its purchase in the face of other options such as multiple separate computers or periods processing. To help ensure attention to performance, we do our own development work on a VAX security kernel system. Performance-critical paths were written in a high-level language and then re-written in assembly language for speed. We have meters to find performance-critical routines, and a rudimentary performance monitor to gather statistics on CPU and I/O usage.

Bug tracking mechanisms are needed both to satisfy NCSC configuration management guidelines, and to give us a means to respond to problems on a timely basis. They also provide a means to check against our definition of quality: having no security bugs and no bug that keeps production work from running. Statistics on the number of bugs and their severity provide concrete feedback on stability.

6.2 Documentation

A real security kernel requires extensive documentation for its users and for its system and security managers. These documents must not only meet the content requirements of the NCSC; they must also be clear and understandable to both novice and sophisticated customers. The VAX security kernel documentation set consists of nine manuals and a reference card. The manuals include a user's guide, guides to both system security and system management, a command reference manual, both basic and advanced programmer's manuals, an installation guide, a master index, and release notes. These manuals have been written to the same quality standards as the manuals for the VMS and ULTRIX-32 operating systems.

7 Comparison with KVM/370

While the VAX security kernel superficially bears a strong resemblance to KVM/370, in that both systems create virtual machines that run at different access classes, the internal structures of the two systems are very different.

Most significantly, KVM/370 was designed as a retrofit to the existing VM/370 product, with a specific goal of leaving at least half of the original code intact [11]. As a result, KVM/370 was structured as shown in Figure 6. The KVM/370 security kernel used a variation on self-virtualization to create a series of NKCPs (Non-Kernel Control Programs), each at a distinct mandatory access class. The NKCPs ran unmodified VM/370 code to create multiple virtual machines that then ran the CMS (Conversational Monitor System), a single-user operating system designed to run in a virtual machine. The disadvantage of this approach is that many functions executed by a virtual machine required two context switches, first into the NKCP and then into the security kernel. By comparison, VAX security kernel achieves a higher performance level by allowing the virtual machines to communicate directly with the security kernel. This makes the VAX security kernel larger than the KVM/370 security kernel, but we believe that the performance gains justify the increase in size.[11]

KVM/370 never implemented support for VMOSs that supported virtual memory. It implemented demand paging within its TCB. By contrast, the VAX security kernel leaves virtual memory support to the VMOSs. As discussed in

[11] This comparison is not strictly fair to KVM/370 because the KVM/370 team was required to maintain compatibility and a large body of original code from VM/370, while the VAX security kernel team had the liberty of designing and coding from scratch.

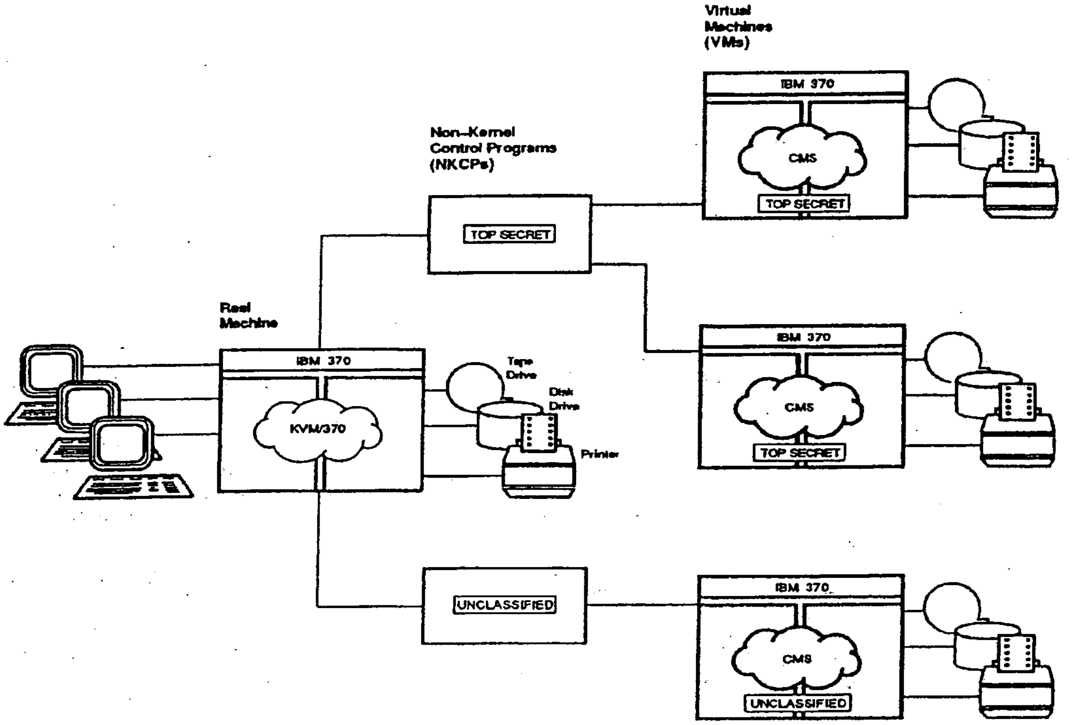

Figure 6: KVM/370 Configuration

Section 3.7, eliminating demand paging reduces kernel complexity and improves performance at the cost of limiting the number of simultaneously active virtual machines.

Another major difference is that KVM/370 has a very limited interface for system management and security management functions. For example, new users cannot be added during online operation. By contrast, the VAX security kernel offers a full complement of system and security management tools, such as are required in a general-purpose system. (See Section 4.)

While performance comparisons are very tricky to make, the relative performance of the VAX security kernel seems better than that of KVM/370. KVM/370 reports [11] performance ranges from 10% to 50% of VM/370, depending on the workload. By contrast, the VAX security kernel exhibits performance ranges from 30% to 90% of VMS capacity, again depending on the workload. The KVM/370 measurements were of an untuned system, while the VAX security kernel measurements were of a system with a limited amount of tuning. The KVM/370 comparisons were to VM/370, itself a virtual-machine monitor with performance degradation compared to a native operating system. The VAX security kernel comparisons were to the native VMS operating system. KVM/370 reported a number of desirable performance optimizations that had not been done, and likewise, we know of a number of optimizations that have not yet been applied to VAX security kernel because of limited development resources.

8 History of the Project

The idea of a virtual-machine monitor security kernel for the VAX, similar in concept to KVM/370, was first conceived by Paul Karger and Steve Lipner in a Mexican restaurant in Palo Alto, CA, immediately after the 1981 Symposium on Security and Privacy. An initial design study [17] concluded in 1982 that such a security kernel would be practical for the VAX architecture.

The security kernel was initially prototyped on a VAX-11/730 system. The VAX 11/730 CPU [34] was particularly attractive because it was vertically microprogrammed, and its microcode was executed from a writeable control store

(WCS) that could be reloaded from magnetic tape cassettes. This environment was ideal for experimenting with alternate microcode extensions to the VAX architecture, although the CPU itself was quite slow.

The VMS operating system first successfully booted in a virtual machine on 19 July 1984. That version of the security kernel was a research prototype and was not a production-quality system. It was extremely slow (due in part to the choice of the VAX-11/730 and in part to the initial software design that emphasized quick development and extensive self-checking, but not performance), and its user interface was extremely crude.

Once the VMM security kernel prototype was running reliably on the VAX-11/730 and we had accomplished some performance tuning (that improved system performance by at least an order of magnitude), we then began investigation of what a production-quality version would be like. The extensions to the VAX architecture were re-implemented on the VAX 8800 family of CPUs to provide a high-performance base for the system. Like the VAX-11/730, the VAX 8800 CPU [24] runs its microcode from a writeable control store (WCS), so modifications were possible. The VAX 8800 microcode is organized horizontally, rather than vertically, and the microcode is pipelined, so the actual implementation of the extensions was much more complex than for the VAX-11/730.

Going from the research prototype to the practical version also gave us the opportunity to revisit a number of design decisions. In particular, the extensions to the VAX architecture to support virtualization were simplified, in part due to the limited availability of microcode memory in the VAX 8800. A performance study of the VAX security kernel prototype revealed that some of our architectural extensions did not provide the expected performance gains, while other extensions would be more valuable. For example, the prototype design included complex microcode assistance for delivering exceptions and interrupts to the virtual machines, but these microcode assists proved not to be useful, and a much simpler scheme was implemented for the VAX 8800. Similarly, performance measurements of the prototype revealed that VAX operating systems (and VMS in particular) use the MTPR instruction to change their interrupt priority level (IPL) much more frequently than anyone had expected. Therefore, the software was changed to optimize this particular path, and microcode assistance was considered, although not implemented in this version.

The move to the production-quality kernel also marked the development of such features as user and system management interfaces, auditing, and error logging. The prototype kernel, as a research kernel, had no need of such tools, but a real A1 system must have them, so that the end users can manage and reliably run real applications on the system.

By January 1988, the kernel was sufficiently stable that some engineers could begin doing their development work on a VM. Also in January 1988, the first VAX security kernel was installed outside the kernel development group. That system was installed in the European ULTRIX Engineering Group in Reading, England for porting the ULTRIX-32 operating system to a virtual machine. ULTRIX 32 first booted in a virtual machine on 15 February 1988, only two months after detailed design for the port began, and less than one month after a working VAX security kernel system was available for use in Reading.

By August 1988, VAX security kernel builds were being done on virtual machines, and by early 1989, essentially all software development work was being done on the kernel. By Spring of 1989, the kernel was sufficiently stable that the VAX 8800 that had been running a conventional VMS time sharing system for the kernel developers was released for other purposes.

9 Conclusions

The VAX security kernel is a working, production-quality VMM security kernel with performance sufficient to support a large number of time-sharing users. It is sufficiently fast and stable that it supports its own development team. It supports vast amounts of existing user software that has been written for both the VMS and the ULTRIX-32 operating systems, and it supports both operating systems running simultaneously on the same CPU. VAX security kernel is currently (as of February 1990) in the Design Analysis Phase with the National Computer Security Center (NCSC) for an A1 rating. As a research project in what is required to build a practical security kernel, it has been a major success.

The development of VAX security kernel has been long and arduous, and we have learned a number of lessons during that time. Performance of a security kernel is extremely important, and getting good performance is very hard. It requires detailed analysis of what portions of the kernel are performance-critical and a willingness to redesign those portions for performance and possibly re-code them in assembly language or to provide microcode performance assistance.

Building the system twice, once as a research prototype and once as a research study of a production-quality system, was extremely valuable. The second time around, we were able to apply some of the performance lessons learned by adjusting our microcode assistance, and we developed the user and management interfaces that are essential in a real system.

Developing a system to A1 standards is very hard work. Some of the A1 requirements can directly conflict with performance and usability goals, and the testing and review requirements are very time consuming. Furthermore, the export controls imposed on A1 systems can seriously reduce the potential market for a system, making it difficult to recover the costs in achieving the A1 rating. On the other hand, the discipline required to meet A1 requirements definitely improves overall software quality and reliability.

10 Acknowledgements

A great many people have been involved in making the VAX security kernel a success, and space does not permit mentioning them all here. The VAX 11/730 prototype was developed by a team of Paul Karger, Andrew Mason, Clifford Kahn, and Sara Thigpen. The VAX-11/730 microcode extensions were done by Timothy Leonard. Other engineers on the project have included Carol Anderson, Dennis Argo, Mark Bokhan, David Butchart, Tom Casey, Georgette Champagne, Ed Childs, Ron Crane, Dennis Elfstrom, John Ferguson, Alison Gabriel-Reilly, Rumi Gonda, Ray Govotski, Ellen Gugel, Judy Hall, Allen Hsu, Wei Hu, Les Kendall, Bill Kindel, Charles Lo, Marie McClintock, Jeff Mclanson, Linda Nasman, Tai-Chun Pan, Steve Pinkoski, Tony Priborsky, John Purretta, Dan Raizen, Paul Robinson, Paul Sawyer, Ken Seiden, Gabriel Shapira, Rod Shepardson, Rich Simon, Steve Stennett, Henry Teng, Tom Tierney, Susie Troop, and Mary Ellen Zurko. Verification work has been done by Chuck Dermody, David Ellis, Russ Marsden, Ray Modeen, David Wittenberg, and John Wray with assistance from Eve Cohen, Theresa Haley, Sue Landauer and Terry Vickers Benzel of Trusted Information Systems and Jeff Thomas of Aerospace Corporation. The technical writers have included Elena Aschkenasi, Doug Bonin, Elizabeth Guth, John Hurst, Bruce Laru, and Pamela Norton. In addition, the contributions of managers, supervisors, field test coordinators, compiler writers, members of the VMS and the European ULTRIX Engineering groups, product managers, customer service systems engineers, marketing people, operations staff, our illustrator, and secretaries have all been critical to the project. Finally, we must thank our team from the National Computer Security Center for their participation throughout the long development effort and the referees for their suggestions for improving this paper.

References

[1] *A Proposed Interpretation of the TCSEC for Virtual Machine Architectures*. Trusted Information Systems, Inc., Glenwood, MD, draft of 31 March 1989.

[2] David E. Bell and Leonard J. LaPadula. *Computer Security Model: Unified Exposition and Multics Interpretation*. Technical Report ESD TR-75-306, The MITRE Corporation, Bedford, MA, HQ Electronic Systems Division, Hanscom AFB, MA, June 1975.

[3] T. A. Berson and G. L. Barksdale, Jr. KSOS - development methodology for a secure operating system. In *AFIPS Conference Proceedings, Volume 48, 1979 National Computer Conference*, pages 365-371, AFIPS Press, Montvale, NJ, 1979.

[4] Kenneth J. Biba. *Integrity Considerations for Secure Computer Systems*. Technical Report ESD-TR-76-372, The MITRE Corporation, Bedford, MA, HQ Electronic Systems Division, Hanscom AFB, MA, April 1977.

[5] Steven Blotcky, Kevin Lynch, and Steven Lipner. SE/VMS: implementing mandatory security in VAX/VMS. In *Proceedings of the 9th National Computer Security Conference*, pages 47-54, National Bureau of Standards, Gaithersburg, MD, 15-18 September 1986.

[6] Lyle A. Cox, Jr. and Roger R. Schell. The structure of a security kernel for a Z8000 multiprocessor. In *Proceedings of the 1981 Symposium on Security and Privacy*, pages 124-129, IEEE Computer Society, Oakland, CA, 27-29 April 1981.

[7] *Department of Defense Trusted Computer System Evaluation Criteria*. DOD 5200.28-STD, Department of Defense, Washington, DC, December 1985.

[8] E. W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341-346, May 1968.

[9] Lester J. Fraim. SCOMP: a solution to the multilevel security problem. *Computer*, 16(7):26-34, July 1983.

[10] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Company, New York, NY, 1988.

[11] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 13-23, IEEE Computer Society, Oakland, CA, 29 April - 2 May 1984.

[12] B. D. Gold, R. R. Linde, R. J. Peeler, M. Schaefer, J. F. Scheid, and P. D. Ward. A security retrofit of VM/370. In *AFIPS Conference Proceedings, Volume 48, 1979 National Computer Conference*, pages 335-344, AFIPS Press, Montvale, NJ, 1979.

[13] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. Ph. D. thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA, February 1973. Published as ESD-TR-73-105, HQ Electronic Systems Division, Hanscom AFB, MA.

[14] *Guide to VMS System Security*. Order No. AA-LA40B-TE. Digital Equipment Corporation, Maynard, MA, June 1989.

[15] Philippe A. Janson. *Using Type Extension to Organize Virtual Memory Mechanisms*. Ph. D. thesis, Department of Electrical Engineering and Computer Science, MIT/LCS/TR-167, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1976.

[16] Paul A. Karger. Computer security research at Digital. In *Proceedings of the Third Seminar on the DoD Computer Security Initiative Program*, pages E-1 - E-6, National Bureau of Standards, Gaithersburg, MD, 18-20 November 1980.

[17] Paul A. Karger. *Preliminary Design of a VAX-11 Virtual Machine Monitor Security Kernel.* Technical Report DEC TR-126, Digital Equipment Corporation, Hudson, MA, 13 January 1982.

[18] Paul A. Karger, Timothy E. Leonard, and Andrew H. Mason. *Computer With Virtual Machine Mode and Multiple Protection Rings.* United States Patent No. 4,787,031, 22 November 1988.

[19] Richard A. Kemmerer. FDM — a specification and verification methodology. In *Proceedings of the Third Seminar on the DoD Computer Security Initiative Program*, pages L-1 - L 19, National Bureau of Standards, Gaithersburg, MD, 18-20 November 1980.

[20] Richard A. Kemmerer. A practical approach to identifying storage and timing channels. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 66-73, IEEE Computer Society, Oakland, CA, 26 - 28 April 1982.

[21] Timothy E. Leonard, editor. *VAX Architecture Reference Manual.* Digital Press, Bedford, MA, 1987.

[22] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security. In *Proceedings of the ACM SIGARCH SIGOPS Workshop on Virtual Computer Systems*, pages 210 224, Harvard University, Cambridge, MA, USA, 26-27 March 1973.

[23] Stuart E. Madnick and John J. Donovan. *Operating Systems.* McGraw-Hill Book Company, New York, NY, 1974.

[24] Sudhindra N. Mishra. The VAX 8800 microarchitecture. *Digital Technical Journal*, (4):20-33, February 1987.

[25] John Nagle. Update on the kernelized security operating system (KSOS). In *Proceedings of the Third Seminar on the DoD Computer Security Initiative Program*, pages Q 1 Q 7, National Bureau of Standards, Gaithersburg, MD, 18-20 November 1980.

[26] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412-421, July 1974.

[27] Gerald J. Popek and Charles S. Kline. The PDP-11 virtual machine architecture: a case study. *Operating Systems Review*, 9(5):97 105, 19 21 November 1975. Proceedings of the Fifth Symposium on Operating Systems Principles, University of Texas, Austin, TX.

[28] David P. Reed. *Processor Multiplexing in a Layered Operating System.* S.M. thesis, Department of Electrical Engineering and Computer Science, MIT/LCS/TR-164, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1976.

[29] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115 123, February 1979.

[30] R. Rhode. *Secure Multilevel Virtual Computer Systems.* Technical Report ESD-TR 74 370, The MITRE Corporation, Bedford, MA, HQ Electronic Systems Division, Hanscom AFB, MA, February 1975.

[31] Kenneth F. Seiden and Jeffrey P. Melanson. The auditing facility for a VMM security kernel. In *1990 IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society, Oakland, CA, 7 9 May 1990.

[32] William R. Shockley, Tien F. Tao, and Michael F. Thompson. An overview of the GEMSOS class A1 technology and application experience. In *Proceedings of the 11th National Computer Security Conference*, pages 238-245, National Bureau of Standards/National Computer Security Center, 17-20 October 1988.

[33] Michael Dennis Vahey. *A Virtualizer Efficiency Device for Virtual Machines.* M. S. thesis, University of California, Los Angeles, CA, 1975.

[34] *VAX-11/730 Central Processing Unit Technical Description.* EK KA730-TD-001, Digital Equipment Corporation, Maynard, MA, May 1982.

[35] *VMS Analyze/Disk_Structure Utility Manual.* Order No. AA LA39A-TE. Digital Equipment Corporation, Maynard, MA, April 1988.

[36] J. Whitmore, A. Bensoussan, P. Green, D. Hunt, A. Kobziar, and J. Stern. *Design for Multics Security Enhancements.* Technical Report ESD-TR-74-176, Honeywell Information Systems, Inc., HQ Electronic Systems Division, Hanscom AFB, MA, December 1973.